SOLID - Dependency Inversion

Programação Orientada a Objeto II

SOLID - Dependency Inversion

Programação Orientada a Objeto II

"Os sistemas mais flexíveis são aqueles em que as dependências de código-fonte se referem apenas a abstrações e não a itens concretos".

Frequentemente, quando vamos projetar nossas próprias classes, precisamos definir composições com outras classes. Porém, projetos mudam de acordo com a evolução dos requisitos ou o comportamento desejado. Acontece que, se a nossa composição estiver muito acoplada, não conseguiremos realizar as substituições necessárias.

O que estamos falando aqui, é do **princípio do baixo acoplamento**, onde a dependência entre as classes da composição é baixa. Conseguimos isso através de abstrações ao invés das classes concretas.

Problemática

Retomando nossa aplicação de secretaria, desejamos sempre que for realizada nova matrícula, que o aluno seja notificado por e-mail.

```
public class MatriculaService {
    private EmailService emailService;
    public MatriculaService() {
        this.emailService = new EmailService();
    }
    public void realizarMatricula(Aluno aluno) {
        //realizar matrícula
        this.emailService.send(aluno.getEmail());
    }
}
```

Problemática

Retomando nossa aplicação de secretaria, desejamos sempre que for realizada nova matrícula, que o aluno seja notificado por e-mail.

```
public class MatriculaService {
    private EmailService emailService;
    public MatriculaService() {
        this.emailService = new EmailService();
    }
    public void realizarMatricula(Aluno aluno) {
        //realizar matrícula
        this.emailService.send(aluno.getEmail());
    }
}
```

Solução

Veja que o principal ponto de acoplamento no código anterior está no construtor da classe **MatriculaService**.

Portanto, a primeira mudança que podemos fazer é passar a instância por parâmetro no construtor.

```
public class MatriculaService {
    private final EmailService emailService;
    public MatriculaService(EmailService emailService) {
        this.emailService = emailService;
    }
    //...
}
```

Solução

Outra mudança que precisamos fazer é criar uma abstração do meio de comunicação, diminuindo o acoplamento entre as classes.

```
public interface ComunicacaoService {
   void send(String mensagem, String destinatario);
}
public class EmailService implements ComunicacaoService {
   //code
}
public class WhatsappService implements ComunicacaoService {
   //code
}
public class SmsService implements ComunicacaoService {
   //code
}
```

SOLID - Dependency Inversion

Programação Orientada a Objeto II

```
public class MatriculaService {
    private final ComunicacaoService comunicacaoService;
    public MatriculaService(ComunicacaoService comunicacaoService) {
        this.comunicacaoService = comunicacaoService;
    }
    public void realizarMatricula(Aluno aluno) {
        //realizar matrícula
        this.comunicacaoService.send("Matrícula realizada.",
aluno.getDestinatario());
    }
}
```

SOLID - Dependency Inversion

Programação Orientada a Objeto II

```
ComunicacaoService comunicacaoService = new EmailService();
MatriculaService matriculaService = new MatriculaService(comunicacaoService);
matriculaService.realizarMatricula(new Aluno());//envia e-mail
```

SOLID - Dependency Inversion

Programação Orientada a Objeto II

Resumindo

- **Classes de alto nível não devem depender de classes de baixo nível. Ambos devem depender de abstrações.**
- **Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.**

Ou seja, em vez de uma classe depender diretamente de outra, ela deve depender de uma interface ou abstração.

Isso torna o código mais flexível e fácil de manter.

Outro exemplo de código

Imagine que temos um sistema de pagamento que depende diretamente de uma classe `PayPalPayment`:

```
class PayPalPayment {
    fun pay(amount: Double) {
        println("Pagamento de $$amount via PayPal realizado!")
    }
}

class PaymentService {
    private val payment = PayPalPayment() // Dependência direta

    fun processPayment(amount: Double) {
        payment.pay(amount)
    }
}
```

SOLID - Dependency Inversion

Programação Orientada a Objeto II

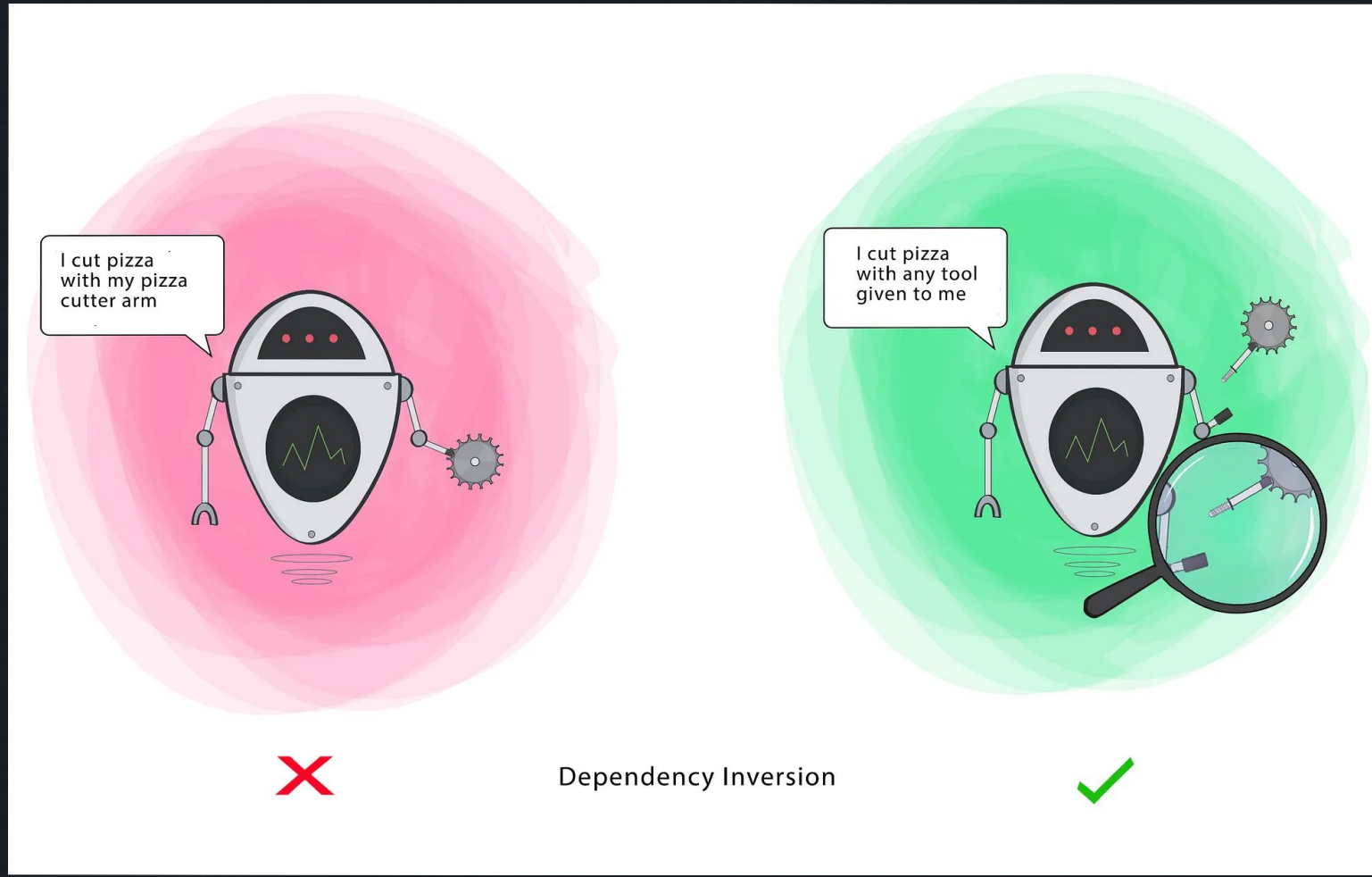
Outro exemplo de código

Vamos criar uma interface para definir um método genérico de pagamento:

```
interface PaymentGateway {
    fun pay(amount: Double)
}

class PayPalPayment : PaymentGateway {
    override fun pay(amount : Double) {
        println("Pagamento de $$amount via PayPal realizado!" )
    }
}

class StripePayment : PaymentGateway {
    override fun pay(amount : Double) {
        println("Pagamento de $$amount via Stripe realizado!" )
    }
}

class PaymentService(private val paymentGateway: PaymentGateway) { // Dependendo de uma
abstração
    fun processPayment(amount: Double) {
        paymentGateway.pay(amount)
    }
}
```

SOLID - Dependency Inversion

Programação Orientada a Objeto II



Obrigada